Faculdade de Engenharia da Universidade do Porto

Conceção e Análise de Algoritmos

Tema 4: FarmFresh2U: empresa de distribuição de frutas e vegetais

10/04/2020

Turma 3 Grupo 09

Participantes:

Ana Teresa Feliciano da Cruz	up201806460@fe.up.pt
André Filipe Meireles do Nascimento	up201806461@fe.up.pt
António Cadilha da Cunha Bezerra	up201806854@fe.up.pt

Índice

Descrição do Tema

A FarmFresh2U, uma **empresa de logística**, realiza a entrega de **cabazes** de **produtos frescos** recolhidos numa **quinta** da região da Guarda nas **casas** dos seus clientes. Os **camiões** da empresa são mantidos numa **garagem**, que pode estar localizada na quinta ou noutro local. No final do circuito de entregas, os camiões regressam à garagem.

Pretende-se implementar um sistema que determine **a melhor rota** para o percurso de entrega dos cabazes. Este deve também realizar a **distribuição** dos mesmos pelos camiões da empresa, de modo a não exceder a capacidade dos camiões e **agrupando** as entregas por zona geográfica para **minimizar** os percursos.

Este problema é bastante semelhante ao **problema do caixeiro-viajante** (Travelling Salesman Problem - TSP), que tenta encontrar o **caminho mínimo** que percorra um dado **conjunto de cidades** e regresse à cidade de origem. Neste caso, não se pretende percorrer cidades mas sim casas de clientes e o ponto final (a garagem) não tem necessáriamente que ser o mesmo que o inicial (a quinta).

No caso em que são usados **vários veículos**, com capacidade **limitada**, é uma instância do *Heterogeneous Fleet Vehicle Routing Problem* (HVRP), que tenta encontrar o **conjunto** de melhores rotas de entrega para uma **frota de veículos** com **diferentes capacidades** máximas.

Serão consideradas **duas etapas** no desenvolvimento da solução.

Etapa 1: Um camião com capacidade ilimitada

Nesta fase considera-se que a empresa tem **apenas um camião** que é capaz de realizar a entrega de **todos os cabazes** a aguardar transporte. O foco será então desenvolver o algoritmo que gere os caminhos mais curtos para realizar todas as entregas e regressar à garagem.

Etapa 2: Vários camiões de diferentes tipos e capacidades

Nesta fase considera-se que a empresa possui **uma frota de camiões** de diferentes capacidades. Será então necessário que o algoritmo, para além de construir o caminho de entrega, **agrupe os cabazes** de modo a não excederem a capacidade de carga máxima dos camiões, de tal forma que os percursos realizados sejam os mais curtos possíveis.

Formalização do problema

Dados de entrada

- Q - Quinta do produtor onde se recolhem os cabazes, caracterizada por:
    - loc - localização geográfica;
- G - Garagem onde os camiões recolhem no final do dia.
- Cab - lista de cabazes para recolha, sendo Cab(i) o i-ésimo elemento, cada um é caracterizado por:
    - peso - peso de cada cabaz;
    - dest - identificação do destinatário;
    - num_f - número de fatura que identifica o cabaz.
- Clientes - lista de clientes, sendo Clientes(i) o i-ésimo elemento. Cada um é caracterizado por:
    - loc - localização geográfica do local de residência;
    - nif - identificação do cliente;
- Cam_i - lista de camiões, sendo Cam_i(i) o seu i-ésimo elemento. Cada um é caracterizado por:
    - cap - capacidade de carga;
    - mat - matrícula do camião;

  (Na etapa 1 considera-se apenas um camião - $|Cam_i| = 1$)
- G_i = (V_i, E_i) - grafo dirigido pesado, composto por:
    - V_i - vértices (localizações geográficas) com:
        - tipo - tipo de localização (quinta, garagem, destinatário, nenhum);
        - Adj - conjunto de arestas que partem do vértice.
    - E_i - arestas (vias de comunicação) com:
        - w - peso da aresta (distância entre os vértices);
        - ID - identificador de uma aresta;
        - dest - vértice de destino.

Dados de saída

- Path - lista ordenada de nós a visitar, cada um caracterizado pelo tipo de nó. Quando são considerados vários camiões, cada é apresentado ainda que camião foi utilizado e a capacidade utilizada.

Restrições

Dados de entrada

- Parâmetros de localização têm que ser localizações geográficas válidas na zona da Guarda;
- O peso de um cabaz tem que ser um número positivo não nulo;
- Todos os cabazes têm que ter um destinatário (dest) que pertença a Clientes;
- num_f é único para cada cabaz;
- nif é único para cada cliente;
- cap_p tem que ser um número positivo não nulo;
- mat é único para cada camião;
- Cada vértice tem de ser de um dos tipos: quinta, garagem, destinatário, nenhum;
- O peso de cada aresta tem que ser um valor positivo não nulo;
- ID é único para cada aresta;
- Arestas não percorríveis não são incluídas no grafo.

Dados de saída

- $\left|Cam_i\right| \leq \left|Cam_f\right|$, não se podem utilizar mais camiões do que os disponíveis;
- O primeiro elemento de Perc tem que pertencer ao conjunto de adjacências da quinta;
- O último elemento de Perc tem que ter a garagem de camião como destino.

Funções objetivo

O objetivo do problema é **minimizar** a **distância percorrida** nos percursos e o **número de camiões** utilizados. Logo, as funções-objetivo a minimizar são:

- $ncam = \left|Cam_f\right|$ (na etapa 2)
- $dist = \sum_{c \in Cam_f} \left[\sum_{e \in Perc} w(e) \right]$

A minimização de *ncam* é prioritária sobre a de *dist*, ou seja, encontrado o número mínimo de camiões que é capaz de realizar todas as entregas, passa-se à otimização do seu percurso, não se analisando percursos que envolvam mais camiões do que o estritamente necessário.

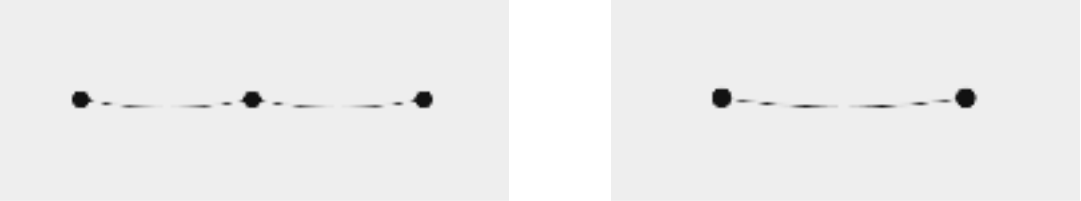
Perspetiva de solução

Técnicas de conceção e Algoritmos a implementar

Pré-processamento do grafo

De modo a simplificar a execução dos algoritmos, o intuito desta etapa é **simplificar o grafo** de modo a excluir **nós irrelevantes**. O critério de decisão terá que ser adequado para evitar que se eliminem nós que pudessem ser úteis e assim se prejudique a qualidade da solução.

Um exemplo de um critério de decisão poderá ser o de eliminar nós intermédios com pelo menos uma aresta de chegada e apenas um nó de destino, como se pode ver no exemplo:



É eliminado o nó intermédio pois não adicionava opções de caminhos diferentes, apenas ligando dois nós que poderiam estar unidos diretamente por uma aresta. Espera-se então conseguir uma diminuição do tamanho do grafo que permita aplicar algoritmos mais rapidamente.

Avaliação da conectividade do grafo

Como os grafos a considerar representam **redes viárias** serão, por natureza, **grafos fortemente conexos**. No entanto, situações pontuais de **estradas cortadas** poderão tornar partes da rede **inacessíveis**.

De modo a não aceitar entregas em zonas inacessíveis do grafo, será feita uma análise das suas componentes fortemente conexas utilizando o **algoritmo de Tarjan**. Este algoritmo encontra os **componentes fortemente conexos** (CFC) de um dado grafo recorrendo à **pesquisa em profundidade** (DFS).

```
algorithm tarjan is
    input: graph G = (V, E)
    output: set of strongly connected components (sets of vertices)

    index := 0
    S := empty stack
    for each v in V do
        if v.index is undefined then
            strongconnect(v)
        end if
    end for
```

```
function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)
    v.onStack := true

    // Consider successors of v
    for each (v, w) in E do
        if w.index is undefined then
            // Successor w has not yet been visited; recurse on it
            strongconnect(w)
            v.lowlink := min(v.lowlink, w.lowlink)
        else if w.onStack then
            // Successor w is in stack S and hence in the current SCC
            // If w is not on stack, then (v, w) is an edge pointing to an SCC already found and must be ignored
            // Note: The next line may look odd - but is correct.
            // It says w.index not w.lowlink; that is deliberate and from the original paper
            v.lowlink := min(v.lowlink, w.index)
        end if
    end for

    // If v is a root node, pop the stack and generate an SCC
    if v.lowlink = v.index then
        start a new strongly connected component
        repeat
            w := S.pop()
            w.onStack := false
            add w to current strongly connected component
        while w ≠ v
        output the current strongly connected component
    end if
end function
```

Encontrados os CFC do grafo, verificar-se-á que todos os pontos de entrega assim como a quinta e a garagem pertencem ao mesmo CFC.

O algoritmo tem complexidade temporal $O(|V|+|E|)$ e espacial $O(|V|)$.

Algoritmos de caminho mais curto

Uma das principais tarefas na resolução deste problema será encontrar o caminho mais curto de um vértice para o outro. O algoritmo que permite calcular estes caminhos é o **algoritmo de Dijkstra**. Existe ainda um outro algoritmo, baseado no de Dijkstra, mas que utiliza uma função de heurística para tentar encontrar soluções mais rapidamente: o **algoritmo A*** (A-Star). Ambos os algoritmos serão implementados para comparar tanto a qualidade das soluções obtidas como o tempo dispendido por cada um.

Algoritmo de Dijkstra

Algoritmo ganancioso que encontra sempre o caminho mais curto entre nós de um grafo pesado, com pesos não negativos. Tem um comportamento semelhante à **pesquisa em largura** (BFS), utilizando uma **fila de prioridade** com mínimo à cabeça para escolher o próximo vértice a ser processado. O algoritmo pode ser adaptado de modo encontrar o caminho mais curto de um nó para todos os outros, ou então entre um par específico de nós. Para tal, é alterada a condição de paragem do algoritmo para que, no caso em que se pretende apenas o caminho entre dois nós em específico, este termine quando o nó no topo da fila de prioridade seja o nó de destino pretendido.

```
1  function Dijkstra(Graph, source):
2      dist[source] ← 0
3
4      create vertex priority queue Q
5
6      for each vertex v in Graph:
7          if v ≠ source
8              dist[v] ← INFINITY
9              prev[v] ← UNDEFINED
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:
15         u ← Q.extract_min()
16         for each neighbor v of u:
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist, prev
```

O pseudo-código apresentado encontra o caminho mais curto desde *source* até todos os outros nós. Na nossa implementação é possível que se venham a considerar ambos os casos (um nó para todos e um para outro) já que poderão ser úteis para aplicações diferentes.

O algoritmo tem complexidade temporal $O((|V|+|E|) * log(|V|)$, que pode ser melhorada para $O(|V| * log(|V|)$ recorrendo a *Fibonacci Heaps*. A complexidade espacial é $O(|V|)$.

Algoritmo A* (A-Star)

Este algoritmo é bastante semelhante ao algoritmo de **Dijkstra**, sendo muitas vezes considerado um "melhoramento" deste último. A melhoria de desempenho é alcançada recorrendo a uma **função de heurística** que "guia" o algoritmo. Assim, consegue encontrar o caminho mais curto analisando um número menor de vértices.

Em vez de escolher o próximo vértice tendo em conta o custo, $g(v,n)$, de percorrer a aresta de v até n, o valor que é minimizado é $f(v,n) = g(v,n) + h(n,f)$, em que $h(n,f)$ é uma função heurística que **estima** o custo de viajar desde o nó n até ao nó objetivo, f. No entanto, para que o algoritmo garanta uma solução **ótima** é necessário que a heurística utilizada seja **admissível**, para tal, deve respeitar a seguinte condição: $\forall n,f \in G,\ w(n,f) \geq h(n,f)$. Ou seja, a heurística nunca pode **sobrestimar** o custo de navegar desde um vértice até ao objetivo.

```
function A-star(Graph, src, dest):
    dist[src] <- h(src, dest)       // h(v, f) is the heuristic function estimating
                                    // the distance between a node, v, and the goal, f
    create Vertex priority queue Q

    for each Vertex v in Graph:
        if v != src:
            dist[v] <- INFINITY
        prev[v] <- UNDEFINED
        Q.add_with_priority(v, dist[v])

    while not_empty(Q):
        v <- Q.pop()

        if equals(v, dest):
            return v

        for each Vertex w in adj(v):
            // g(v, w) is the actual cost of travelling from node v to w
            f = dist[v] - h(v, dest) + g(v, w) + h(w, dest)
            if dist[w] > f:
                dist[w] <- f
                prev[w] <- v
                Q.decrease_priority(w, f)

    return null
```

O pseudo-código aqui apresentado assume também que a função heurística utilizada é uma **heurística consistente**. Quer isto dizer que, para qualquer nó, n, e qualquer nó, w, atingível a partir de n por um caminho a, o custo estimado de chegar ao destino a partir de n, $h(n,f)$, não é maior do que o custo real de chegar a w por a, $g(n,a,w)$, somado ao custo estimado de chegar ao destino a partir de w, $h(w,f)$. Ou seja: $\forall\ a,\ \forall\ n,w,f \in G,\ h(n,f) \leq g(n,a,w) + h(w,f)$. Caso a heurística utilizada seja **admissível mas não consistente** é necessário alterar o algoritmo de modo a permitir revisitar nós já explorados anteriormente.

Dado que o grafo representa um mapa de uma rede viária, a heurística que nos parece mais adequada é a **distância euclidiana** entre os vértices a considerar, dada por: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Esta função é **admissível e consistente**.

Em termos de complexidade, no caso em que é utilizada uma heurística não informativa, o algoritmo é idêntico ao de **Dijkstra**: complexidade temporal $O((|V| + |E|) * log(|V|)$ e espacial $O(|V|)$. Na prática, a complexidade está dependente da qualidade da heurística utilizada.

Pesquisa bidirecional

A **pesquisa bidirecional** consiste em tentar encontrar o caminho mais curto num grafo realizando **duas pesquisas em simultâneo**: uma **pesquisa normal** a partir do nó de origem e uma **pesquisa inversa** a partir do nó destino. A pesquisa termina quando os subgrafos gerados por cada pesquisa se **intersetam**. Esta estratégia pode **reduzir** significativamente o **número de explorações** necessárias.

Considerando um grafo com fator de ramificação b e distância desde o nó origem ao nó destino de d, a complexidade de uma pesquisa **tradicional** é $O(b^d)$. Recorrendo à pesquisa **bidirecional** a complexidade é de $O(b^{d/2})$ para cada pesquisa, ou seja, $O(b^{d/2} * 2)$ no total, o que é uma redução bastante significativa.

Pretende-se implementar algoritmos de pesquisa bidirecional ao algoritmo de **Dijkstra** e **A*** de modo a avaliar possíveis ganhos de desempenho obtidos.

Construção de rotas simples (TSP)

Como mencionado na descrição do tema, a primeira etapa da implementação será focada na resolução do problema tendo em conta **apenas um veículo** que é capaz de satisfazer **todas as entregas**. Desta forma, o objetivo será, dados um vértice inicial q (que representa a quinta), um conjunto de i pontos de entrega p_i e um vértice final g (que representa a garagem), encontrar um caminho que vá de q a g, passando por p_i. Admite-se que a **ordem** pela qual os pontos de entrega serão visitados deve ser determinada pelo algoritmo. Trata-se, pois, de uma instância do ***Travelling Salesman Problem*** (TSP).

A solução ótima deste problema é extremamente difícil de calcular. A solução por ***brute-force*** envolve testar todas as permutações de pontos possíveis, tendo complexidade $O(n!)$. O crescimento fatorial faz com que, na prática, se recorram a **métodos heurísticos** que permitam conseguir **aproximações** da solução ótima, mas em tempo computacional muito menor. Existe uma grande variedade de heurísticas para aproximar soluções do TSP, de seguida apresentar-se-ão as que serão consideradas para implementação.

Vizinho mais próximo - *Nearest Neighbour* (NN)

Um dos algoritmos mais simples e rápidos para resolver o TSP é o algoritmo de vizinho mais próximo. Como o nome indica, este algoritmo escolhe como próximo nó a visitar aquele que seja o mais próximo do nó atual. Como medida de proximidade pode ser utilizado tanto o custo real de viajar de um nó para o outro como uma aproximação do mesmo com uma função heurística.

Assumindo que todos os pontos a visitar estão no mesmo componente fortemente conexo, o algoritmo pode ser concebido com base neste pseudo-código.

```
1  // Vi - vértice origem, Vf - vértice destino,
2  // POIs - lista de vértices a visitar
3  // ord - ordem de visita, inicialmente vazia
4  function NN-Search(Vi, Vf, POIs, ord <- {}):
5      ord.insert(Vi)
6
7      if equals(Vi, Vf):
8          return ord
9
10     next <- find_nearest(Vi, POIs) // Função que devolve o nó mais próximo
11     POIs.remove(next)
12
13     NN-Search(next, Vf, POIs, ord)
```

A qualidade das soluções obtidas com este algoritmo, no entanto, não é muito alta. Ao escolher o vértice imediatamente mais próximo, perdem-se oportunidades de ganhos futuros. Existem algoritmos de melhoria de soluções que permitem melhorar a qualidade de uma solução obtida pelo NN, como o algoritmo 2-opt que será apresentado mais à frente. Ainda assim, há alterações que é possível fazer ao próprio algoritmo de vizinho mais próximo que tentam melhorar as hipóteses de escolher ordens mais eficientes.

Vizinho mais próximo aleatório - *Randomized NN* (RNN)

Uma dessas variações é o algoritmo do vizinho mais próximo aleatório. Este algoritmo escolhe o próximo vértice aleatoriamente a partir de um conjunto dos n vizinhos mais próximos. Apesar de ser uma solução rudimentar, a introdução da aleatoriedade aumenta a probabilidade de se escolherem percursos mais eficientes.

```
// Vi - vértice origem, Vf - vértice destino,
// POIs - lista de vértices a visitar
// ord - ordem de visita, inicialmente vazia
// n - número de vértices a considerar para escolha aleatória
function RNN-Search(Vi, Vf, POIs, ord <- {}, n):
    ord.insert(Vi)

    if equals(Vi, Vf):
        return ord

    next_set <- find_n_nearest(Vi, POIs, n) // Função que devolve os n nós mais próximos
    next <- next_set.random()     // Função que devolve um elemento qualquer do conjunto
    POIs.remove(next)

    RNN-Search(next, Vf, POIs, ord, n)
```

Como o algoritmo pode devolver resultados diferentes de cada vez que é utilizado, pode ser corrido várias vezes permitindo explorar rapidamente várias soluções. Tal como o algoritmo NN, as soluções obtidas pelo RNN podem ser melhoradas com o algoritmo 2-opt que se apresenta de seguida.

Algoritmo 2-opt

O algoritmo 2-opt é um algoritmo de pesquisa local, ou seja, tenta encontrar soluções na vizinhança da solução obtida que melhorem o seu desempenho. O princípio subjacente ao seu funcionamento é tentar evitar caminhos que se cruzam, trocando a ordem pelos quais os vértices são visitados.

A troca é efetuada da seguinte maneira:

```
procedure 2optSwap(route, i, k) {
    1. take route[0] to route[i-1] and add them in order to new_route
    2. take route[i] to route[k] and add them in reverse order to new_route
    3. take route[k+1] to end and add them in order to new_route
    return new_route;
}
```

E o funcionamento geral do algoritmo é este:

```
repeat until no improvement is made {
    start_again:
    best_distance = calculateTotalDistance(existing_route)
    for (i = 1; i <= number of nodes eligible to be swapped - 1; i++) {
        for (k = i + 1; k <= number of nodes eligible to be swapped; k++) {
            new_route = 2optSwap(existing_route, i, k)
            new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
                existing_route = new_route
                best_distance = new_distance
                goto start_again
            }
        }
    }
}
```

Em que se pode controlar o número de nós que é possível trocar, de modo a evitar trocas com os nós fixos (o inicial e o final, neste caso).

Combinando este algoritmo com os previamente mencionados, é possível obter aproximações relativamente boas da solução ótima com um gasto computacional relativamente baixo. Na nossa implementação iremos escolher a combinação que dê resultados de melhor qualidade num tempo razoável.

Construção de rotas com frota heterogénea (HVRP)

Já na segunda etapa de desenvolvimento, o problema a solucionar será o de encontrar as rotas mais económicas para as entregas considerando que a empresa tem uma frota de camiões de diferentes tipos e capacidades. Esta é uma instância do *Vehicle Routing Problem* (VRP), com a restrição de que os veículos a considerar têm capacidades diferentes uns dos outros. Por vezes, o problema com esta restrição é chamado *Heterogeneous VRP* (HVRP).

Tal como o TSP, a solução ótima para este problema é encontrada testando exaustivamente todas as combinações possíveis de veículos a usar e rotas que estes podem seguir, tendo complexidade exponencial. Deste modo, recorre-se também a heurísticas para aproximar soluções que possam ser obtidas em tempo útil.

As heurísticas tradicionalmente utilizadas para resolver este problema podem ser classificadas como um de três tipos. Heurísticas construtivas, que constroem aos poucos uma solução, e heurísticas de duas fases, que dividem o problema em duas componentes: a de agrupar vértices para formar rotas possíveis e a de construir os caminhos das rotas em si. As heurísticas de duas fases podem ser divididas em dois tipos: *cluster-first*, *route-second* ou *route-first*, *cluster-second*.

Algoritmo *Sweep*

Este é um algoritmo do tipo *cluster-first*, *route-second*. Os grupos de vértices são formados por "varrimento" dos pontos a visitar centrado no nó inicial, considerando uma representação dos vértices em coordenadas polares (θ_i, ρ_i). O *clustering* tem em conta a capacidade máxima dos veículos disponíveis e pode ainda ser imposto um limite ao tamanho da rota, de modo a não serem formados grupos inviáveis ou demasiado extensos. Os passos do algoritmo tradicional são:

1. Escolher um veículo k não utilizado.
2. Começando a partir do vértice com menor ângulo em relação ao vértice inicial, atribuir vértices à rota do veículo k até ser atingida a sua capacidade ou, eventualmente, um tamanho máximo de comprimento da rota.
3. Repetir a partir de 1. caso ainda existam vértices por percorrer.
4. Resolver uma instância do TSP para cada uma das rotas atribuídas a cada veículo.

Como este algoritmo foi concebido para um VRP com frota homogénea, será necessário realizar algumas adaptações para a sua aplicação a uma frota heterogénea. Nomeadamente, na escolha do próximo veículo a utilizar, escolher sempre o veículo com maior capacidade possível, de modo a diminuir o número de veículos utilizados.

É de ressalvar que nada garante que o *clustering* realizado pelo algoritmo seja viável. Mesmo que a capacidade da frota seja superior à capacidade necessária para realizar as entregas, é possível que o agrupamento das mesmas resulte numa situação impossível. Uma maneira de corrigir isso poderá ser começar o varrimento a partir de um vértice distinto de modo a gerar outras rotas. A deteção e solução destes *edge-cases* será algo a ter em conta durante a implementação.

Funcionalidades e casos de utilização

A aplicação a implementar utilizará uma interface simplista, interagindo com o utilizador através de um sistema de menus a partir dos quais pode selecionar diversas opções:

- Carregamento e pré-processamento do grafo - de modo a permitir selecionar grafos de diferente dimensão e complexidade, útil para as fases de teste;
- Visualização do grafo carregado (recorrendo ao GraphViewer);
- Gestão de cabazes, com criação de novos cabazes, edição dos já existentes ou até mesmo cancelamento de encomendas;
- Gestão da frota de veículos, recorrendo à integração de novos veículos de diversos tipos e capacidades na frota ou remoção de veículos já existentes;
- Cálculo de melhores rotas, com possibilidade de seleção dos algoritmos a utilizar e se se considera apenas um veículo ou uma frota de veículos;
- Visualização da rota calculada.

Conclusão preliminar

Neste relatório foi feita uma exploração do tema proposto bem como a conceção de uma possível estratégia de resolução do problema tendo por base os algoritmos explorados na Unidade Curricular bem como algoritmos e ideias fruto de uma pesquisa mais aprofundada. Essa pesquisa teve como objetivo relacionar o tema com problemas conhecidos cujas estratégias de solução estivessem bem documentadas.

A elaboração do relatório em si foi desafiante pois o tema se revelou consideravelmente mais complexo do que parecia à primeira vista. Assim, para que a solução encontrada tivesse uma qualidade aceitável, foi necessário um esforço maior de pesquisa e comparação de algoritmos do que inicialmente esperado.

Apesar de tudo, consideramos que o relatório é um importante instrumento de planeamento para a posterior implementação. Ainda assim, foi um desafio tentar idealizar as estratégias sem as testar.

A pesquisa e redação do relatório foram distribuídas igualmente pelos elementos do grupo, sendo frequentemente efetuada em conjunto com partilha imediata de ideias.

Casos de utilização implementados

A interface inicial do programa é a seguinte:

Este é também o aspeto geral de todos os menus, que são implementados recorrendo à classe *Menu.h*.

Funcionalidades

- **Load graph:** permite carregar grafos a partir dos ficheiros na pasta *maps*. Alguns contêm ficheiros com dados relacionados com o contexto do problema, que permitem a realização de rotas para entregas, com vários camiões, etc.
- **Show graph:** mostra o grafo carregado numa janela à parte (recorrendo à API *GraphViewer*).
- **Pathfind:** permite encontrar o caminho mais curto entre dois nós, selecionando qual o algoritmo *ShortestPath* que pretende utilizar (Dijkstra ou A*, unidirecional ou bidirecional). No final é apresentado o tamanho do caminho encontrado e o tempo de execução do algoritmo, e ainda a opção para visualizar o caminho (recorrendo ao *GraphViewer* ou na consola).
- **Delivery path:** permite construir um caminho mais curto que passe por certos pontos de interesse (POI) fornecidos.
    - *'Make delivery route'*: realiza todas as entregas considerando um camião com capacidade infinita.
    - *'Make delivery route with different trucks'*: realiza as entregas com a frota de camiões fornecida.
    - 'General TSP': pede ao utilizador o nó inicial, nó final, e os POIs que quer visitar, permitindo assim resolver TSP em qualquer grafo, mesmo sem contexto.

Após a seleção de uma destas opções, é pedido que utilizador que escolha qual algoritmo usar (NN ou RNN, com ou sem 2-opt), e de seguida apresentado o tamanho da rota encontrada, o tempo de execução do algoritmo, e a opção para mostrar o caminho recorrendo ao GraphViewer ou na consola.

- **Connectivity:** apresenta a opção para visualizar o maior componente fortemente conexo (CFC, em inglês SCC) recorrendo ao algoritmo de Tarjan. Existe também a opção de criar um grafo a partir do maior CFC para posteriormente se poder trabalhar sobre ele com os restantes algoritmos que precisam de um grafo conexo.
- **Performance:** permite realizar testes de performance dos algoritmos implementados, mostrando algumas estatísticas sobre os tempos de execução dos algoritmos e em certos casos gerando amostras aleatórias.
- **Farm Management:** permite a gestão de clientes, cabazes e camiões quando é carregado um grafo com contexto.

Algoritmos Implementados

Algoritmo de Tarjan

Algoritmo implementado conforme o pseudocódigo apresentado na **perspetiva de solução**. O algoritmo permitiu caracterizar os mapas fornecidos em termos da sua **conectividade**, identificando quantos componentes fortemente conexos existiam em cada um, permitindo ao utilizador visualizar o componente de maior dimensão e passar a considerar apenas esse componente como o grafo sobre o qual os algoritmos serão aplicados.

Análise de conectividade

As imagens apresentam o resultado obtido pelo algoritmo de Tarjan aplicado ao mapa *PortoFull*, bem como a visualização do seu maior componente fortemente conexo.

```
--------------------
Connectivity analysis
--------------------
Total number of strongly connected components: 13709
Largest SCC size: 26098
Average SCC size: 3.91137
```

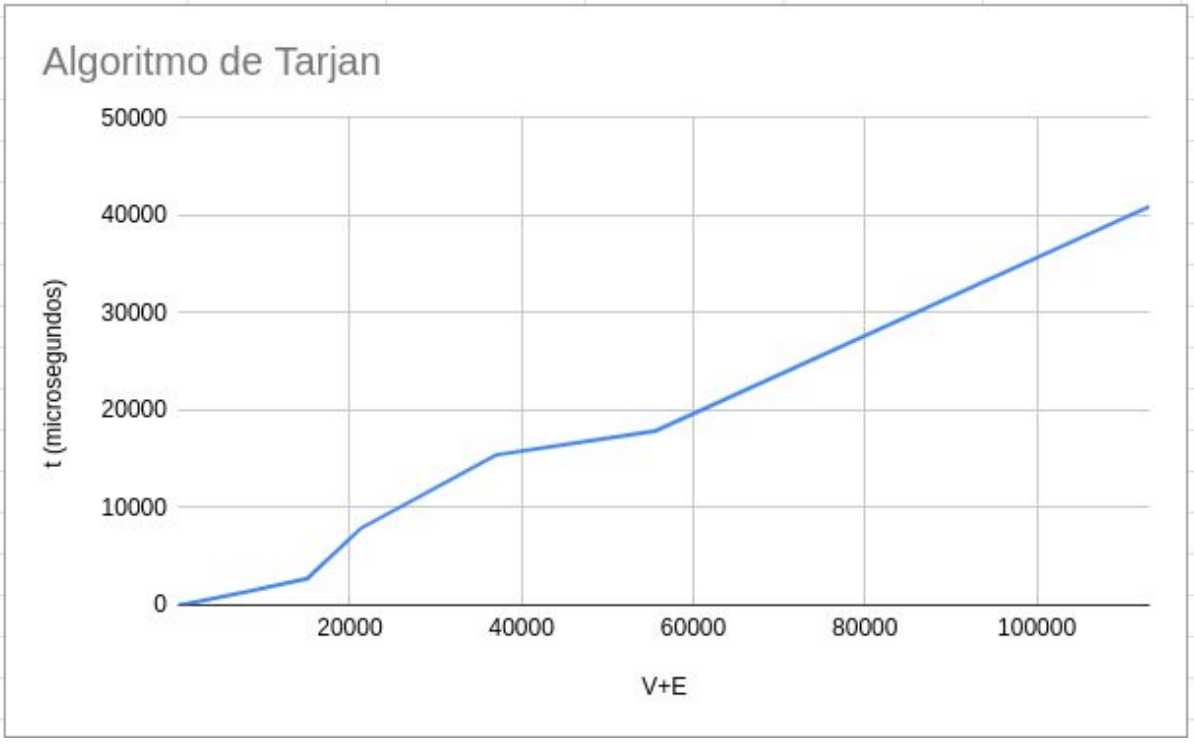
Complexidade

Teórica:

- Temporal: $O(|V|+|E|)$
- Espacial: $O(|V|)$

Empírica:

	V+E	Min	Max	Avg	N
4x4	65	10	84	18,06	1000
8x8	225	28	118	39,213	1000
16x16	833	97	296	109,863	1000
Penafiel Strong	8201	1246	8228	1394,47	1000
Espinho Strong	15046	2259	7520	2720,06	1000
Penafiel Full	21281	6592	17050	7 878,66	1000
Espinho Full	37032	13719	24496	15420,8	1000
Porto Strong	55586	16097	29384	17875,1	1000
Porto Full	113147	38322	56345	40926,8	1000



Como se pode observar no gráfico, a complexidade temporal é linear em V+E.

Algoritmos de Dijkstra e A*

Estes algoritmos de caminho mais curto foram implementados conforme o apresentado no pseudocódigo apresentado nas secções anteriores (**Dijkstra**, **A***). Dado que os algoritmos têm a mesma lógica subjacente, a sua implementação foi

feita utilizando uma função comum ***genericShortestPath*** (em Graph.h/.cpp), que recebe uma função heurística como parâmetro: uma heurística constante de valor 0 no caso do algoritmo de Dijkstra e a função de distância euclidiana no caso do algoritmo A*.

Algoritmos bi-direcionais

Para além dos algoritmos unidirecionais, foram também implementadas as respetivas variantes bidirecionais. O pseudocódigo relativo a esta implementação pode ser consultado no **Anexo I**. Tal como no caso unidirecional, a função que implementa estes algoritmos é comum a tanto o algoritmo de Dijkstra como A*, variando apenas a função heurística utilizada. Como se analisa de seguida, estes algoritmos foram capazes de melhorar a performance de certos casos.

Complexidade

Teórica:

- Temporal: $O((|V|+|E|) * log(|V|))$
- Espacial: $O(|V|)$

Empírica:

	V+E	Dijkstra	Dijkstra BiDir	A*	A* BiDir	N
4x4	65	12,294	20,3	13,038	25,508	500
8x8	225	48,726	67,048	16,985	33,356	500
16x16	833	137,432	172,892	68,37	93,834	500
Penafiel Strong	8201	1102,73	1619,06	864,056	1385,86	500
Espinho Strong	15046	2368,9	2039,33	1188,53	1287,01	500
Porto Strong	55586	11607,2	9848,02	4954,32	3512,85	500

Como esperado, o algoritmo A* apresenta melhor performance do que o de Dijkstra, e as respetivas variantes bidirecionais melhoram o desempenho nalguns casos. Observamos que o melhoramento não foi tão acentuado como previsto em grafos de grelha, pois a performance destes algoritmos para além de condicionada pela dimensão dos grafos, também é influenciada pelas suas características.

Algoritmos *Nearest-Neighbor* e *Randomized-NN*

Algoritmos implementados conforme o pseudocódigo apresentado nas secções anteriores (**NN**, **RNN**). A implementação usada calcula a proximidade real de um nó para o outro recorrendo a um algoritmo de caminho mais curto e não uma aproximação heurística da mesma. Aumenta a complexidade e tempo de execução mas dado o contexto ser uma rede viária, em que os pontos mais próximos podem com frequência não ser os mais facilmente acessíveis, optamos por fazer esse compromisso.

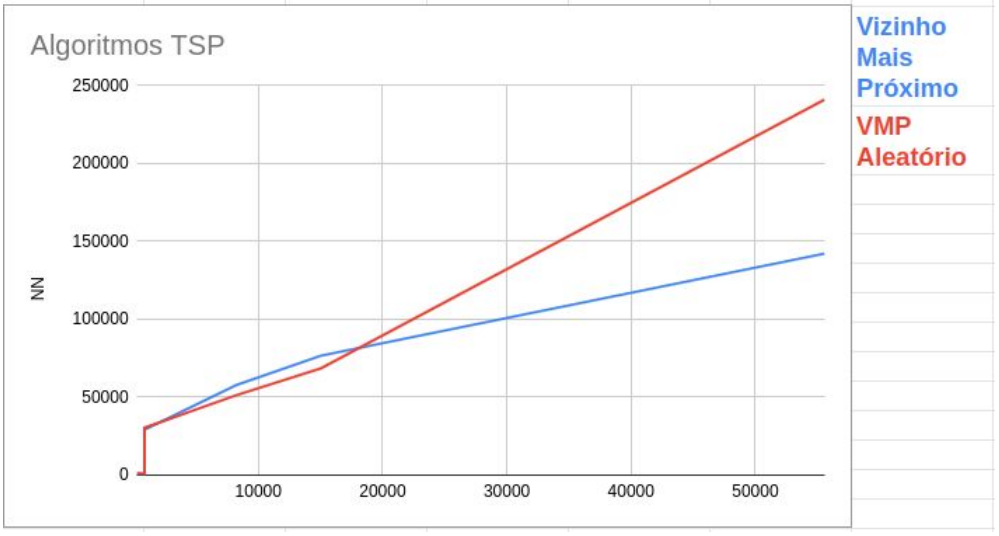
Complexidade

Teórica:

- Temporal: $O((|V|+|E|) * log(|V|) * n^2)$, para n pontos de interesse
- Espacial: $O(|V| * n)$

Empírica:

	V+E	nPOI	NN	RNN	N
4x4	65	10	575,72	566,66	100
8x8	225	10	900,65	1045,65	100
16x16	833	5	696,11	902,85	100
16x16	833	10	3684,31	3072,14	100
16x16	833	20	12245,5	13610,4	100
16x16	833	30	29110,7	30376,2	100
Penafiel Strong	8201	10	57699,8	51180,7	100
Espinho Strong	15046	10	76636,6	68557,1	100
Porto Strong	55586	10	142172	241025	100



Algoritmo 2-opt

Algoritmo implementado conforme o pseudocódigo apresentado na **perspetiva de solução**. O critério de paragem é diferente consoante o tamanho do problema, pois como o caminho é recalculado a cada troca, em grafos de grandes dimensões e considerando muitos pontos de entrega, o tempo de execução era demasiado elevado.

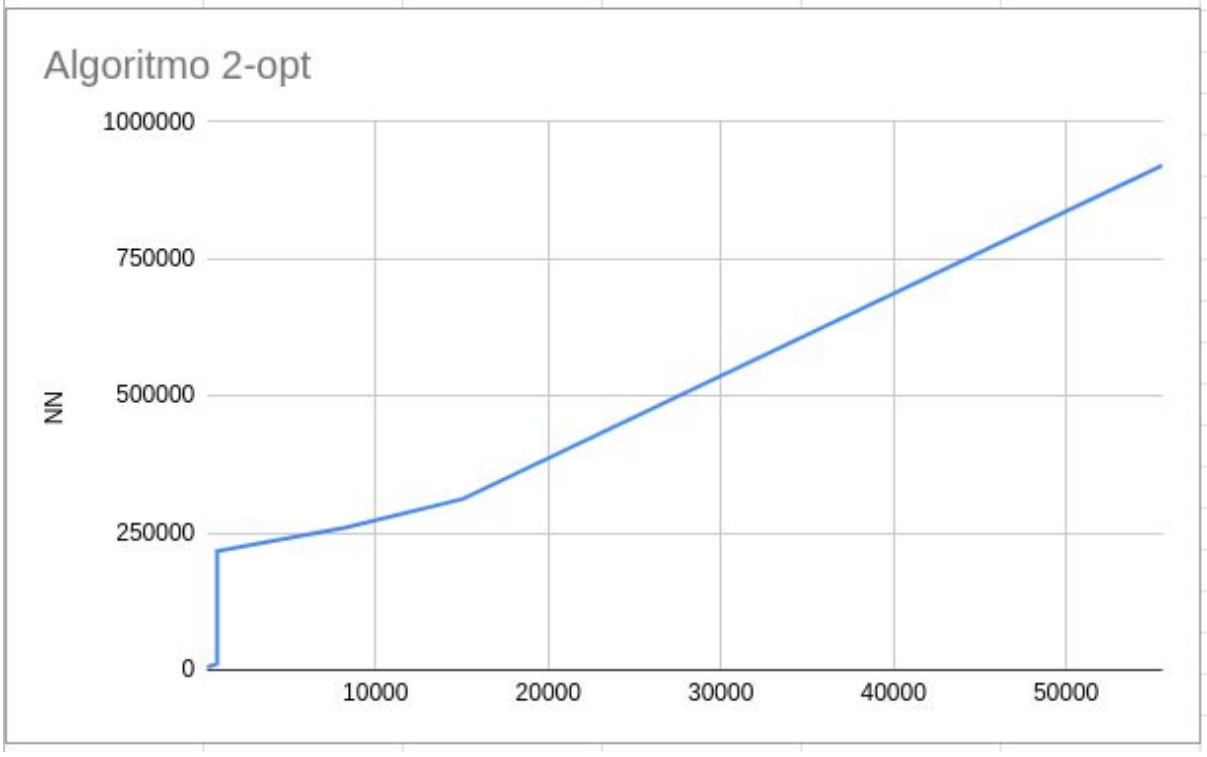
Complexidade

Teórica:

- Temporal: $O((|V|+|E|) * log(|V|) * n^2)$
- Espacial: $O(|V| * n)$

Empírica:

	V+E	nPOI	NN+2-opt	N
4x4	65	5	2520,95	100
8x8	225	5	5351,24	100
16x16	833	5	11109,6	100
16x16	833	10	98978	100
16x16	833	20	96255,9	100
16x16	833	30	216825	100
Penafiel Strong	8201	5	259075	100
Espinho Strong	15046	5	311796	100
Porto Strong	55586	5	920609	100



Algoritmo *Sweep*

Algoritmo implementado conforme o pseudocódigo apresentado na **perspetiva de solução**. Para evitar casos em que não é encontrada uma solução possível, o algoritmo recorre à seguinte estratégia:

1. Tentar novamente fazendo o varrimento no sentido inverso.
2. Se não encontrar solução, fazer a distribuição do camião mais pequeno para o maior, considerando varrimentos no sentido horário e anti-horário.
3. Se não encontrar solução, fazer o varrimento começando num dos pontos de interesse.
4. Se não encontrar solução, tentar 1 e 2
5. Se não encontrar solução, repetir a partir de 3 até percorrer todos os pontos de interesse.
6. Se continuar a não ser encontrada uma solução, o algoritmo termina.

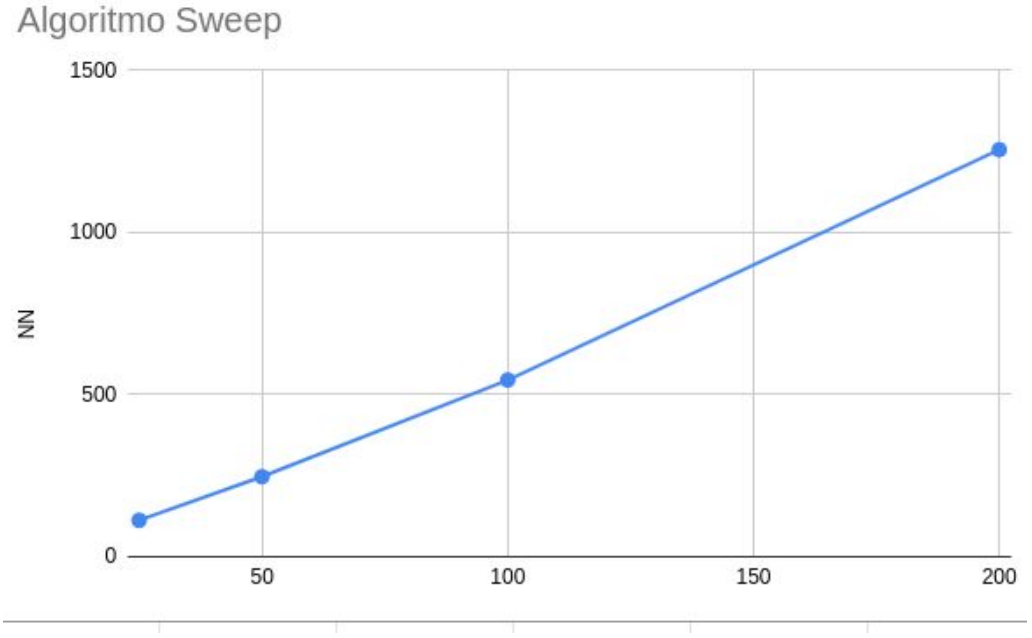
Complexidade

Teórica:

- Temporal: $O(n * log(n))$, para n POIs (caso médio de std::sort)
- Espacial: $O(n)$

Empírica:

	V+E	nPOI	Sweep	N
6x16	833	10	37,374	500
6x16	833	25	112,322	500
6x16	833	50	246,996	500
6x16	833	100	545,106	500
6x17	833	200	1255,66	500



Estruturas de Dados

Graph

A classe *graph* guarda um vetor de apontadores de vértices, cada um dos quais tem: um identificador, um vetor de arestas com origem no vértice (*outgoing*) e um vetor de arestas que vão dar ao vértice (*incoming*). Guardar os dois tipos de arestas permite facilmente percorrer o grafo inverso, sem necessidade de o calcular explicitamente. Para além do vetor, usamos um mapa para associar o id de um vértice à sua posição no vetor para acelerar a pesquisa.

Os algoritmos de grafos foram implementados como métodos da classe e adicionadas variáveis auxiliares aos vértices para os mesmos.

MutablePriorityQueue

Implementação adaptada da fornecida nas aulas práticas, para suportar a criação de duas filas em simultâneo, uma para o grafo normal e outra percorrendo o grafo em sentido inverso.

Path

Estrutura de dados que permite guardar um vetor que representa um caminho no grafo e o seu comprimento. Tem métodos para juntar dois caminhos que concatenam os vetores e adicionam os pesos de cada caminho.

Route

Estrutura de dados que permite guardar um vetor com os pontos de entrega de uma rota específica, cada um com o seu peso, e o peso total de todos os pontos.

Farm

Guarda toda a informação necessária para resolução de problemas com contexto, nomeadamente o *nodeID* da quinta e da garagem, um vetor com a frota de camiões disponíveis, um mapa para os clientes e um mapa com um vetor de todos os cabazes de um certo cliente (ambas keys são o nif do cliente para acelerar a pesquisa). Guarda ainda o nome dos ficheiros de dados e variáveis que permitem saber se os dados foram alterados em tempo de execução.

Conclusão

A análise prévia do problema realizada para a primeira entrega foi bastante útil, uma vez que a seleção dos algoritmos foi adequada e fomos capazes de os implementar.

Inicialmente tivemos muitas dificuldades com o *Graph Viewer,* o que nos atrasou no desenvolvimento do projeto. No entanto, conseguimos superar estas dificuldades e continuar o seu desenvolvimento.

A *interface* implementada é bastante simples e direta. Para o contexto achamos que é completa o suficiente, apesar de ter algumas limitações.

Os testes empíricos que foram realizados poderiam ser mais extensos, mas dada a diversidade de algoritmos implementados, para que tal fosse possível iria exigir um esforço muito mais significativo. Dado que este não era o foco principal do trabalho, este tempo foi canalizado para desenvolvimento de outras partes.

Um dos problemas que também afetou a realização deste projeto foi o facto de só termos acesso aos mapas reais na semana da entrega. Deste modo, não nos apercebemos de certas funcionalidades ou melhorias que poderiam ter sido implementadas mais cedo.

Em suma, apesar das dificuldades encontradas, conseguimos superar e concluir o projeto orgulhosamente. Foram implementadas todos os algoritmos e funcionalidades pensadas com uma boa estrutura e organização de código e, por isso, estamos satisfeitos com o nosso projeto.

- O trabalho nesta etapa foi novamente distribuído de forma equitativa e justa de modo que, a percentagem deve ser distribuída igualmente pelos três.
    - 33% Ana Teresa
    - 33% André Nascimento
    - 33% António Bezerra

Bibliografia

- Apresentações das aulas teóricas de Concepção e Análise de Algoritmos 2019/2020
- Tarjan's strongly connected components algorithm, https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
- Dijkstra's algorithm, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- Fibonacci Heap | Set 1 (Introduction), https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/
- A* search algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm
- A* Search Algorithm, https://www.geeksforgeeks.org/a-search-algorithm/
- Bidirectional Search, https://www.geeksforgeeks.org/bidirectional-search/
- Travelling salesman problem, https://en.wikipedia.org/wiki/Travelling_salesman_problem
- The basics of search algorithms explained with intuitive visualisations, https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03b8552
- 2-opt, https://en.wikipedia.org/wiki/2-opt
- Russell, S., Norvig, P., (1994) *Artificial Intelligence: A modern approach*. Prentice Hall
- Routing a Heterogeneous Fleet of Vehicles, https://www.researchgate.net/publication/226187312_Routing_a_Heterogeneous_Fleet_of_Vehicles
- Vehicle routing problem, https://en.wikipedia.org/wiki/Vehicle_routing_problem
- Toth, P., Vigo, D., (2002). *The Vehicle Routing Problem*. SIAM

Anexos

Anexo I

```
genericBiDirShortestPath(src, dest, h) {
    // Set-up starting values for source/destination vertices
    src->dist = h(src->getInfo(), dest->getInfo());
    dest->dist_inv = h(dest->getInfo(), src->getInfo());

    // Insert values into forward and reverse priority queue
    fwdQ.insert(src);
    revQ.insert(dest);

    // Alternating search loop
    while (!fwdQ.empty() && !revQ.empty()) {
        // Forward search
        fwdQ.extractMin();
        fwdVisited.insert(fwdV);
        if (revVisited.contains(fwdV)) {
            mid = fwdV;
            break;
        }
        // Explore edges in regular graph
        relaxFwdEdges(fwdV);

        // Reverse search
        revV = revQ.extractMin();
        revVisited.insert(revV);
        if (fwdVisited.contains(revV)) {
            mid = revV;
            break;
        }
        // Explore edges in reverse graph -> doesn't need to be explicitly
        // computed since nodes store incoming edges
        relaxRevEdges(revV);
    }
```

```
    // Path join
    // Best length so far
    length = mid->dist - h(mid->getInfo(), dest->getInfo())
             + mid->dist_inv - h(mid->getInfo(), src->getInfo());

    // Verify if alternative paths (not through the mid vertex) are better
    for (node_a in fwdVisited) {
        for (edge : node_a->outgoing) {
            node_b = edge->dest;
            if (revVisited.contains(node_b)) {
                new_length = node_a->dist - h(node_a->getInfo(), dest->getInfo())
                             + node_b->dist_inv - h(node_b->getInfo(), src->getInfo())
                             + edge->cost;
                if (temp_length < length) {
                    node_b->path = node_a;
                    mid = node_b;
                    length = temp_length;
                }
            }
        }
    }

    buildPath();
}
```